Int	::=	the domain of (unbounded) integer numbers, with usual operations on them
Bool	::=	the domain of Booleans
Id	::=	standard identifiers
AExp	::=	*Int*
	|	*Id*
	|	*AExp* + *AExp*
	|	*AExp* / *AExp*
BExp	::=	*Bool*
	|	*AExp* <= *AExp*
	|	! *BExp*
	|	*BExp* && *BExp*
Block	::=	{}
	|	{ *Stmt* }
Stmt	::=	*Block*
	|	*Id* = *AExp* ;
	|	*Stmt* *Stmt*
	|	`if` (*BExp*) *Block* `else` *Block*
	|	`while` (*BExp*) *Block*
Pgm	::=	`int` **List{***Id***}** ; *Stmt*

Figure 3.1: Syntax of IMP, a small imperative language, using algebraic BNF.

3.1 IMP: A Simple Imperative Language

To illustrate the various semantic styles discussed in this chapter, we have chosen a small imperative language, called IMP, whose syntax is inspired from C and Java. In fact, if we wrap an IMP program in a `main(){...}` function the we get a valid C program. IMP has arithmetic expressions which include the domain of arbitrarily large integer numbers, Boolean expressions, assignment statements, conditional statements, while loop statements, and sequential composition of statements. Statements can be grouped in blocks surrounded with curly brackets, and the branches of the conditional and the loop body are required to be blocks. All variables used in an IMP program need to be declared at the beginning of the program, can only hold integer values (for simplicity, IMP has no Boolean variables), and are initialized with default value 0.

3.1.1 IMP Syntax

We here define the syntax of IMP, first using the Backus-Naur form (BNF) notation for context-free grammars and then using the alternative and completely equivalent mixfix algebraic notation (see Section 2.1.3). The latter is in general more appropriate for semantic developments of a language.

IMP Syntax as a Context-Free Grammar

Figure 3.1 shows the syntax of IMP using the algebraic BNF notation. In this book we implicitly assume parentheses as part of any syntax, without defining them explicitly. Parentheses can be freely used for grouping, to increase clarity and/or to avoid ambiguity in parsing. For example, with the syntax in Figure 3.1, `(x + 3) / y` is a well-formed IMP arithmetic expression.

The only algebraic feature in the IMP syntax in Figure 3.1 is the use of **List{***Id* **}** for variable declarations (last production), which in this case is clear: one can declare a comma-separated list of variables. To stay more conventional in notation, we refrained from replacing the productions *Stmt* ::= {} | *Stmt Stmt* with the algebraic production *Stmt* ::= $\mathbf{List}^{\{\}}_{\square}${*Stmt*} which captures the idea of statement sequentialization more naturally. Moreover, our syntax for statement sequential composition allows ambiguous parsing. Indeed, if $s_1, s_2, s_3 \in$ *Stmt* then $s_1\ s_2\ s_3$ can be parsed either as $(s_1\ s_2)\ s_3$ or as $s_1\ (s_2\ s_3)$. However, the semantics of statement sequential composition will be such that the parsing ambiguity is irrelevant (but that may not always be the case). It may be worthwhile pointing out that one should not get tricked by thinking that different parsings mean different evaluation orders. In our case here, both $(s_1\ s_2)\ s_3$ and $s_1\ (s_2\ s_3)$ will proceed by evaluating the three statements in order. The difference between the two is that the former will first evaluate $s_1\ s_2$ and then s_3, while the latter will first evaluate s_1 and then $s_2\ s_3$; in either case, s_1, s_2 and s_3 will end up being evaluated in the same order: first s_1, then s_2, and then s_3.

The IMP language constructs have their usual imperative meaning. For diversity and demonstration purposes, when giving the various semantics of IMP we will assume that + is *non-deterministic* (it evaluates the two subexpressions in any order, possibly interleaving their corresponding evaluation steps), / is non-deterministic and *partial* (it will stuck the program when a division by zero takes place), <= is *left-right sequential* (it first evaluates the left subexpression and then the right subexpression), and that && is left-right sequential and *short-circuited* (it first evaluates the left subexpression and then it conditionally evaluates the right only if the left evaluated to true).

One of the main reasons for which arithmetic language constructs like + above are allowed to be non-deterministic in language semantic definitions is because one wants to allow flexibility in how the language is implemented, and not because these operations are indeed intended to have fully non-deterministic, or random, behaviors in all implementations. In other words, their non-determinism is to a large extent an artifact of their intended *underspecification*. Some language manuals actually state explicitly that one should not rely on the order in which the arguments of language constructs are evaluated. In practice, it is considered to be programmers' responsibility to write their programs in such a way that one does not get different behaviors when the arguments are evaluated in different orders.

To better understand the existing semantic approaches and to expose some of their limitations, Section 3.5 discusses extensions of IMP with expression side effects (a variable increment operation), with abrupt termination (a halt statement), with dynamic threads and join synchronization, with local variable declarations, as well as with all of these together; the resulting language is called IMP++. The extension with side effects, in particular, makes the evaluation strategies of + , <= and && semantically relevant.

Each semantical approach relies on some basic mathematical infrastructure, such as integers, Booleans, etc., because each semantic definition reduces the semantics of the language constructs to those domains. We will assume available any needed mathematical domains, as well as basic operations on them which are clearly tagged (e.g., $+_{Int}$ for the addition of integer numbers, etc.) to distinguish them from homonymous operations which are language constructs. Unless otherwise stated, we assume no implementation-specific restrictions in our mathematical domains; for example, we assume integer numbers to be arbitrarily large rather than representable on 32 bits, etc. We can think of the underlying domains used in language semantics as parameters of the semantics; indeed, changing the meaning of these domains changes the meaning of all language semantics using them. We also assume that each mathematical domain is endowed with a special element, written $\perp$ for all domains to avoid notational clutter, corresponding to *undefined* values of that domain. Some of these mathematical domains are defined in Chapter 2; appropriate references will be given when such domains are used.

sorts:

Int, $Bool$, Id, $AExp$, $BExp$, $Block$, $Stmt$, Pgm

subsorts:

$Int, Id < AExp$

$Bool < BExp$

$Block < Stmt$

operations:

$$\begin{array}{rcl}
_ + _ & : & AExp \times AExp \to AExp \\
_ / _ & : & AExp \times AExp \to AExp \\
_ <= _ & : & AExp \times AExp \to BExp \\
! _ & : & BExp \to BExp \\
_ \&\& _ & : & BExp \times BExp \to BExp \\
\{\} & : & \to Block \\
\{ _ \} & : & Stmt \to Block \\
_ = _ ; & : & Id \times AExp \to Stmt \\
_ _ & : & Stmt \times Stmt \to Stmt \\
\texttt{if}\,(_)\,_\,\texttt{else}\,_ & : & BExp \times Block \times Block \to Stmt \\
\texttt{while}\,(_)\,_ & : & BExp \times Block \to Stmt \\
\texttt{int}\,_;\,_ & : & \mathbf{List}\{Id\} \times Stmt \to Pgm
\end{array}$$

Figure 3.2: Syntax of IMP as an algebraic signature.

We take the freedom to tacitly use the following naming conventions for meta or mathematical variables[1] ranging over IMP-specific terms throughout the remainder of this chapter: $x, X \in Id$; $a, A \in AExp$; $b, B \in BExp$; $s, S \in Stmt$; $i, I \in Int$; $t, T \in Bool$; $p, P \in Pgm$. Any of these can be primed or indexed.

IMP Syntax as an Algebraic Signature

Following the relationship between the CFG and the mixfix algebraic notations explained in Section 2.1.3, the BNF syntax in Figure 3.1 can be associated the entirely equivalent algebraic signature in Figure 3.2 with one (mixfix) operation per production: the terminals mixed with underscores form the name of the operation and the non-terminals give its arity. This signature is easy to define in any rewrite engine or theorem prover; moreover, it can also be defined as a data-type or corresponding structure in any programming language. We next show how it can be defined in Maude.

★ Definition of IMP Syntax in Maude

Using the Maude notation for algebraic signatures, the algebraic signature in Figure 3.2 can yield the Maude syntax module in Figure 3.3. We have additionally picked some appropriate precedences and formatting attributes for the various language syntactic constructs (see Section 2.5.6 for more details on Maude and the meaning of these attributes).

[1] Recall that we use an *italic* font for such variables, in contrast to the `typewriter` font that we use for code (including program variable identifiers, integers, operation symbols, etc.). For example, if we write $x, \texttt{x} \in Id$ then we mean an arbitrary identifier that x refers to, and *the concrete* identifier `x`. The latter can appear in programs, while the former cannot. The former is mainly used to define semantics or state properties of the language.

```
mod IMP-SYNTAX is including PL-INT + PL-BOOL + PL-ID .
--- AExp
  sort AExp .  subsorts Int Id < AExp .
  op _+_ : AExp AExp -> AExp [prec 33 gather (E e) format (d b o d)] .
  op _/_ : AExp AExp -> AExp [prec 31 gather (E e) format (d b o d)] .
--- BExp
  sort BExp .  subsort Bool < BExp .
  op _<=_ : AExp AExp -> BExp  [prec 37 format (d b o d)] .
  op !_ : BExp -> BExp [prec 53 format (b o d)] .
  op _&&_ : BExp BExp -> BExp [prec 55 gather (E e) format (d b o d)] .
--- Block and Stmt
  sorts Block Stmt .  subsort Block < Stmt .
  op {} : -> Block [format (b b o)] .
  op {_} : Stmt -> Block [format (d n++i n--i d)] .
  op _=_; : Id AExp -> Stmt [prec 40 format (d b o b o)] .
  op __ : Stmt Stmt -> Stmt [prec 60 gather (e E) format (d ni d)] .
  op if(_)_else_ : BExp Block Block -> Stmt [prec 59 format (b so d d s nib o d)] .
  op while(_)_ : BExp Block -> Stmt [prec 59 format (b so d d s d)] .
--- Pgm
  sort Pgm .
  op int_;_ : List{Id} Stmt -> Pgm [prec 70 format (nb o d ni d)] .
endm
```

Figure 3.3: IMP syntax as an algebraic signature in Maude. This definition assumes appropriate modules PL-INT, PL-BOOL and PL-ID defining corresponding sorts Int, Bool, and Id, respectively.

The module IMP-SYNTAX in Figure 3.3 imports three builtin modules, namely: PL-INT, which we assume it provides a sort Int; PL-BOOL, which we assume provides a sort Bool; and PL-ID which we assume provides a sort Id. We do not give the precise definitions of these modules here, particularly because one may have many different ways to define them. In our examples from here on in the rest of this chapter we assume that PL-INT contains all the integer numbers as constants of sort Int, that PL-BOOL contains the constants true and false of sort Bool, and that PL-ID contains all the letters in the alphabet as constants of sort Id. Also, we assume that the module PL-INT comes equipped with as many builtin operations on integers as needed. To avoid operator name conflicts caused by Maude's operator overloading capabilities, we urge the reader to *not* use the Maude builtin INT and BOOL modules, but instead to overwrite them. Appendix A.1 shows one possible way to do this: we define new modules PL-INT and PL-BOOL "hooked" to the builtin integer and Boolean values but defining only a subset of operations on them and with clearly tagged names to avoid name overloading, e.g., _+Int_, _/Int_, etc.

Recall from Sections 2.4.6 and 2.5.6 that lists, sets, bags, and maps are trivial algebraic structures which can be easily defined in Maude; consequently, we take the freedom to use them without definition whenever needed, as we did with using the sort List{Id} in Figure 3.3.

To test the syntax, one can now parse various IMP programs, such as:

```
Maude> parse
   int n, s ;
   n = 100 ;
   while (!(n <= 0)) {
     s = s + n ;
     n = n + -1 ;
   }
.
```

```
mod IMP-PROGRAMS is including IMP-SYNTAX .
  ops sumPgm collatzPgm countPrimesPgm : -> Pgm .
  ops collatzStmt multiplicationStmt primalityStmt : -> Stmt .
  eq sumPgm = (
     int n, s ;
     n = 100 ;
     while (!(n <= 0)) {
       s = s + n ;
       n = n + -1 ;
     }    ) .

  eq collatzStmt = (
     while (!(n <= 1)) {
       s = s + 1 ;  q = n / 2 ;  r = q + q + 1 ;
       if (r <= n) { n = n + n + n + 1 ; } else { n = q ; }
     }     ) .

  eq collatzPgm = (
     int m, n, q, r, s ;
     m = 10 ;
     while (!(m <= 2)) {
       n = m ;
       m = m + -1 ;
       collatzStmt
     }     ) .

  eq multiplicationStmt = (        --- fast multiplication (base 2) algorithm
     z = 0 ;
     while (!(x <= 0)) {
       q = x / 2 ;
       r = q + q + 1 ;
       if (r <= x) { z = z + y ; } else {}
       x = q ;
       y = y + y ;
     }     ) .

  eq primalityStmt = (
     i = 2 ;  q = n / i ;  t = 1 ;
     while (i <= q && 1 <= t) {
       x = i ;
       y = q ;
       multiplicationStmt
       if (n <= z) { t = 0 ; } else { i = i + 1 ;  q = n / i ; }
     }     ) .

  eq countPrimesPgm = (
     int i, m, n, q, r, s, t, x, y, z ;
     m = 10 ;  n = 2 ;
     while (n <= m) {
       primalityStmt
       if (1 <= t) { s = s + 1 ; } else {}
       n = n + 1 ;
     }     ) .
endm
```

Figure 3.4: IMP programs defined in a Maude module IMP-PROGRAMS.

Now it is a good time to define a module, say `IMP-PROGRAMS`, containing as many IMP programs as one bears to write. Figure 3.4 shows such a module containing several IMP programs. Note that we took advantage of Maude's rewriting capabilities to save space and reuse some of the defined fragments of programs as "macros". The program `sumPgm` calculates the sum of numbers from 1 to 100; since we do not have subtraction in IMP, we decremented the value of `n` by adding -1.

The program `collatzPgm` in Figure 3.4 tests Collatz' conjecture for all numbers from 1 to 10, counting the total number of steps in `s`. The Collatz conjecture, still unsolved, is named after Lothar Collatz (but also known as the $3n+1$ conjecture), who first proposed it in 1937. Take any natural number n. If n is even, divide it by 2 to get $n/2$, if n is odd multiply it by 3 and add 1 to obtain $3n+1$. Repeat the process indefinitely. The conjecture claims that no matter what number you start with, you will always eventually reach 1. Paul Erdös said about the Collatz conjecture: "Mathematics is not yet ready for such problems." While we do not attempt to solve it, we can test it even in a simple language like IMP. It is a good example program to test IMP semantics because it makes use of almost all IMP's language constructs and also has nested loops. The macro `collatzStmt` detaches the check of a single `n` from the top-level loop iterating `n` through all $2 < \texttt{n} \leq \texttt{m}$. Note that, since we do not have multiplication and test for even numbers in IMP, we mimic them using the existing IMP constructs.

Finally, the program `countPrimesPgm` counts all the prime numbers up to `m`. It uses `primalityStmt`, which checks whether `n` is prime or not (writing `t` to 1 or to 0, respectively), and `primalityStmt` makes use of `multiplicationStmt`, which implements a fast base 2 multiplication algorithm. Defining such a module with programs helps us to test the desired language syntax (Maude will report errors if the programs that appear in the right-hand-sides of the equations are not parsable), and will also help us later on to test the various semantics that we will define.

3.1.2 IMP State

Any operational semantics of IMP needs some appropriate notion of *state*, which is expected to map program variables to integer values. Moreover, since IMP disallows uses of undeclared variables, it suffices for the state of a given program to only map the declared variables to integer values and stay undefined in the variables which were not declared.

Fortunately, all these desired IMP state operations correspond to conventional mathematical operations on *partial finite-domain functions* from variables to integers in $[Id \rightharpoonup Int]^{finite}$ (see Section 2.1.2) or, equivalently, to structures of sort $\mathbf{Map}\{Id \mapsto Int\}$ defined using equations (see Section 2.4.6 for details on the notation and the equivalence); we let *State* be an alias for the map sort above. From a semantic point of view, the equations defining such map structures are computationally invisible: semantic transitions that are part of various IMP semantics will be performed *modulo* these equations. In other words, state lookup and update operations will not count as computational steps, so they will not interfere with or undesirably modify the intended computational granularity of the defined language.

We let σ, σ', σ_1, etc., range over states. By defining IMP states as partial finite-domain functions $\sigma : Id \rightharpoonup Int$, we have a very natural notion of undefinedness for a variable that has not been declared and thus has not been initialized in a state: variable x is considered *undefined* in a state σ if and only if $x \notin Dom(\sigma)$. We may use the terminology *state lookup* for the operation $\_(\_) : State \times Id \rightharpoonup Int$, the terminology *state update* for the operation $\_[\_/\_] : State \times Int \times Id \to State$, and the terminology *state initialization* for the operation $\_ \mapsto \_ : \mathbf{List}\{Id\} \times Int \to State$.

Recall from Section 2.1.2 that the lookup operation is itself a partial function, because the state to lookup may be undefined in the variable of interest; as usual, we let $\bot$ denote the undefined state and we write as expected $\sigma(x) = \bot$ and $\sigma(x) \neq \bot$ when the state σ is undefined and, respectively, defined in variable x. Recall

```
mod STATE is including PL-INT + PL-ID .
  sort State .

  op _|->_ : List{Id} Int -> State [prec 0] .
  op .State : -> State .
  op _&_ : State State -> State [assoc comm id: .State format(d s s d)] .

  op _(_) : State Id -> Int [prec 0] .          --- lookup
  op _[_/_] : State Int Id -> State [prec 0] .  --- update

  var Sigma : State .  var I I' : Int .  var X X' : Id .  var Xl : List{Id} .

  eq X |-> undefined = .State .                 --- "undefine" a variable in a state

  eq (Sigma & X |-> I)(X) = I .
  eq Sigma(X) = undefined [owise] .

  eq (Sigma &  X |-> I)[I' / X ] = (Sigma & X |-> I') .
  eq Sigma[I / X] = (Sigma & X |-> I) [owise] .

  eq (X,X',Xl) |-> I = X |-> I & X' |-> I & Xl |-> I .
  eq .List{Id} |-> I = .State .
endm
```

Figure 3.5: The IMP state defined in Maude.

also from Section 2.1.2 that the update operation can be used not only to update maps but also to "undefine" particular elements in their domain: $\sigma[\bot/x]$ is the same as σ in all elements different from x and is undefined in x. Finally, recall also from Section 2.1.2 that the initialization operation yields a partial function mapping each element in the first list argument to the element given as second argument. These can be easily defined equationally, following the equational approach to partial finite-domain functions in Section 2.4.6.

★ Definition of IMP State in Maude

Figure 3.5 adapts the generic Maude definition of partial finite-domain functions in Section 2.5.6 for our purpose here: the generic sorts `Source` and `Target` are replaced by `Id` and `Int`, respectively. Recall from Section 2.5.6 that the constant `undefined` has sort `Undefined`, which is a subsort of all sorts corresponding to mathematical domains (e.g., `Int`, `Bool`, etc.). This way, identifiers can be made "undefined" in a state by simply updating them with `undefined` (see the equation dissolving undefined bindings in Figure 3.5).

To avoid overloading the comma "`,`" construct for too many purposes (which particularly may confuse Maude's parser), we took the freedom to rename the associative and commutative construct for states to `&`. The only reason for which we bother to give this obvious module here is because we want the various subsequent semantics of the IMP language, all of them including the module `STATE` in Figure 3.5, to be self-contained and executable in Maude by simply executing all the Maude code in the figures in this chapter.

3.1.3 Notes

The style that we follow in this chapter, namely to pick a simple language and then demonstrate the various language definitional approaches by means of that simple language, is quite common. In fact, we named our language IMP after a similar language introduced by Winskel in his book [87], also called IMP, which

is essentially identical to ours except that it uses a slightly different syntax and does not have variable declarations. For example, Winskel's IMP uses "`:=`" for assignment and "`;`" as statement separator instead of statement terminator, while our IMP's syntax resembles that of common languages like C and Java. Also, since most imperative languages do have variable declarations, we feel it is instructive to include them in our simple language. Winskell gives his IMP a big-step SOS, a small-step SOS, a denotational semantics, and an axiomatic semantics. Later, Nipkow [55] formalized all these semantics of IMP in the Isabelle/HOL proof assistant [56], and used it to formally relate the various semantics, effectively mechanizing most of Winskel's paper proofs; in doing so, Nipkow [55] found several minor errors in Winskel's proofs, thus showing the benefits of mechanization.

Vardejo and Martì-Oliet [83, 84] show how to use Maude to implement executable semantics for several languages following both big-step and small-step SOS approaches. Like us, they also demonstrate how to define different semantics for the same simple language using different styles; they do so both for an imperative language (very similar to our IMP) and for a functional language. Șerbănuță *et al.* [74] use a similar simple imperative language to also demonstrate how to use rewrite logic to define executable semantics. In fact, this chapter is an extension of [74], both in breadth and in depth. For example, we state and prove general faithful rewrite logic representation results for each of the semantic approaches, while [74] did the same only for the particular simple imperative language considered there. Also, we cover new approaches here, such as denotational semantics, which were not covered in [83, 84, 74].